SMART CONTRACT AUDIT REPORT

for

Atoll

Prepared By: Xiaomi Huang

PeckShield

December 3, 2024

Document Properties

Client	Atoll
Title	Smart Contract Audit Report
Target	Atoll
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 3, 2024	Xuxian Jiang	Final Release
1.0-rc	December 1, 2024	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Atoll` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Atoll

`Atoll` protocol builds upon `Frax`'s `AMO` concept, using `AMO` as the main component to create a novel multi-peg mechanism. This mechanism serves as a `liquidity router` connecting various pegging token (`LSD`/`LRT`) products, amplifying market fluctuations to generate higher yields. Liquidity provider (`LP`) participants only need to engage in farming, while the protocol's `AMO` automatically executes complex buying and selling strategies, distributing profits into the `LP` pool. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Atoll

Item	Description
Name	Atoll
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 3, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Marvin051499/atoll-smart-contracts.git (7748359)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Marvin051499/atoll-smart-contracts.git (51f232e)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Atoll` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Allowance Management Upon Liquidity Update	Coding Practices	Resolved
PVE-002	Low	Necessity of Single-Shot Address Configuration	Init. and Cleanup	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Allowance Management Upon Liquidity Update

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VelodromeDex/CLPAdapter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Atoll` protocol has two built-in adapters to interact with external `Velodrome`-based `DEX` engines. Thus, it has a constant need of swapping one token to another. It also needs to efficiently manage the allowance that has been permitted to supported routers.

If we use the `VelodromeDexAdapter` as an example, the protocol provides functions to add/remove liquidity into/from a given pool. In the following, we show the code snippets from related routines, i.e., `addLiquidity()`. As the name indicates, this function is used to add liquidity into the intended pool. It comes to our attention that it can be improved by resetting the token allowance to zero at the end, namely `IERC20(stableCoin).safeApprove(veloRouter, 0)` and `IERC20(pegCoin).safeApprove(veloRouter, 0)`.

```
    function addLiquidity(uint256 _amountPeg, uint256 _amountStable, uint256
        _minAmountLP) external override onlyAMO onlyCalm {
        // 0 - input validation
        require(_amountPeg > 0 && _amountStable > 0, "Invalid Amounts");
        require(IERC20(pegCoin).balanceOf(address(this)) >= _amountPeg, "No Enough
            PegCoin");
        require(IERC20(stableCoin).balanceOf(address(this)) >= _amountStable, "No Enough
            StableCoin");
        uint256 _minAmountPeg = (_amountPeg * addLiquiditySlippage) / ONE;
        uint256 _minAmountStable = (_amountStable * addLiquiditySlippage) / ONE;

        // 1 - approve
        IERC20(stableCoin).safeIncreaseAllowance(veloRouter, _amountStable);
```

```
        IERC20(pegCoin).safeIncreaseAllowance(veloRouter, _amountPeg);

        // 2 - perform add liquidity
        IVelodromeRouter(veloRouter).addLiquidity(
            pegCoin,
            stableCoin,
            isStable,
            _amountPeg,
            _amountStable,
            _minAmountPeg,
            _minAmountStable,
            address(this),
            block.timestamp
        );

        // 3 - deposit LP tokens to gauge
        uint256 balLP = IERC20(veloPair).balanceOf(address(this));
        IERC20(veloPair).safeIncreaseAllowance(veloGauge, balLP);
        IVelodromeGauge(veloGauge).deposit(balLP, address(this));
        _transferPegAndStableToAMO();
        // After return, AMO will check the received amount of LP tokens
    }
```

Listing 3.1: VelodromeDexAdapter::addLiquidity()

Note that another routine with the same name in `VelodromeCLPAdapter` shares a similar issue.

Recommendation Remove any remaining allowance after the actual swap operation.

Status This issue has been fixed in the following commit: `51f232e`.

3.2 Necessity of Single-Shot Address Configuration

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Initialization and Cleanup [6]
- CWE subcategory: CWE-1188 [2]

Description

The `Atoll` protocol has a number of contracts and many of them have a `configAddress()` function which is used to set up a number of key parameters. Using the `VelodromeDexAdapter` contract as an example, its `configAddress()` function is used to configure a number of contract addresses, including `AMO`, `pegCoin`, `stableCoin` and `veloGauge`. To facilitate our discussion, we show below the related code snippet.

```
    function configAddress(
        address _AMO,
        address _pegCoin,
        address _stableCoin,
        address _veloPair,
        address _veloGauge,
        address _veloRouter,
        address _veloFactory,
        address[] memory _veloRewardToken
    ) external onlyOwner {
        AMO = _AMO;
        pegCoin = _pegCoin;
        stableCoin = _stableCoin;
        veloPair = _veloPair;
        veloGauge = _veloGauge;
        veloRouter = _veloRouter;
        veloFactory = _veloFactory;
        veloRewardToken = new address[](_veloRewardToken.length);
        for (uint256 i = 0; i < _veloRewardToken.length; i++) {
            veloRewardToken[i] = _veloRewardToken[i];
        }

        address token0 = ISolidlyPair(veloPair).token0();
        address token1 = ISolidlyPair(veloPair).token1();
        if (token0 == pegCoin) {
            pegIsZero = true;
            require(token1 == stableCoin, "Invalid Pair");
        } else if (token1 == pegCoin) {
            pegIsZero = false;
            require(token0 == stableCoin, "Invalid Pair");
        } else {
            revert("Invalid Pair");
        }

        uint8 decimalsPeg = ERC20(pegCoin).decimals();
        uint8 decimalsStable = ERC20(stableCoin).decimals();
        decimalDiff = 10 ** (decimalsPeg - decimalsStable);
    }
```

Listing 3.2: `VelodromeDexAdapter::configAddress()`

Apparently the above logic only ensures the caller is authenticated and allowed by the system. But it does not provide the guarantee that the `configAddress()` function can be called only once. Considering multiple initializations could cause unexpected errors for the contract's execution, we strongly suggest to make sure `configAddress()` could only be called once.

Recommendation Consider the need of ensuring that the `configAddress()` function could only be called once during the entire lifetime.

Status This issue has been fixed in the following commit: `51f232e`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In the `Atoll` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, pause/unpause protocol, and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function addToBlacklist(address account) public {
        require(msg.sender == owner(), "Only the contract owner can add to the blacklist");
        require(!_blacklist[account], "Account is already blacklisted");
        _blacklist[account] = true;
        emit BlacklistAdded(account);
    }
    ...
    function removeFromBlacklist(address account) public {
        require(msg.sender == owner(), "Only the contract owner can remove from the blacklist");
        require(_blacklist[account], "Account is not blacklisted");
        _blacklist[account] = false;
        emit BlacklistRemoved(account);
    }
    ...
    function rescue(address target, uint256 value, bytes calldata data) external onlyOwner {
        (bool success, ) = target.call{value: value}(data);
        require(success, "Rescue: Call failed");
    }
    ...
    function RescueTokenToOwner(address token) external onlyOwner {
        if (IERC20(token).balanceOf(address(this)) > 0) {
            IERC20(token).safeTransfer(owner(), IERC20(token).balanceOf(address(this)));
        }
    }
```

Listing 3.3: Example Privileged Functions in `StakedToken`

Note that if the privileged `owner` account is a plain `EOA` account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Atoll` protocol, which builds upon `Frax`'s `AMO` concept, using `AMO` as the main component to create a novel multi-peg mechanism. This mechanism serves as a `liquidity router` connecting various pegging token (`LSD/LRT`) products, amplifying market fluctuations to generate higher yields. Liquidity provider (`LP`) participants only need to engage in farming, while the protocol's `AMO` automatically executes complex buying and selling strategies, distributing profits into the `LP` pool. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.